AD-A039 916   BOEING AEROSPACE CO   SEATTLE WASH   F/G 9/2

SOFTWARE ERROR DATA ACQUISITION.(U)

APR 77   M J FRIES   F30602-76-C-0152

UNCLASSIFIED   RADC-TR-77-130   NL

1 OF 1

AD
A039916

END

DATE
FILMED
6-77

RADC-TR-77-130
Final Technical Report
April 1977

SOFTWARE ERROR DATA ACQUISITION

Boeing Aerospace Company

[illegible]

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED: Alan N. Sukert

ALAN N. SUKERT, Captain, USAF
Project Engineer

APPROVED: Robert D. Krutz

ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-77-130	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE ERROR DATA ACQUISITION.		5. TYPE OF REPORT & PERIOD COVERED Final ~~Technical Report~~, Feb [redacted] - Nov 76,
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) M. J./Fries		8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0152 New
9. PERFORMING ORGANIZATION NAME AND ADDRESS Boeing Aerospace Company P. O. Box 3999 Seattle WA 98124		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ~~62702F~~ ~~55811414~~ 14
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE Apr[redacted]77
		13. NUMBER OF PAGES ~~60~~ 62 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Captain Alan N. Sukert (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Error Categorization
Software Reliability
Software Data Collection
Software Error Categories
Software Data Analysis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Software error data was collected from a large DOD system development project. The errors were analyzed and put into a predefined set of categories. As part of the effort, the times to find and fix the errors were calculated, and the phase of the development project in which the errors arose was determined. Study results were also compared to results of a similar type of study performed by a second contractor who performed analysis of data from another DOD software project.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4 over

059 610 BMc

This report contains a description of the hardware and software systems, the software development process, and the types of data available. Also included are descriptions of the method of categorization and the derivation of other contractually required data items. Finally, discussions are presented concerning: an interpretation of the software error categories, comments on the difficulties and successes in performing the error data collection, an analysis of the data collected by software function, study results, examination of the data by development phase, and recommendations for future software data collection studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY OF ACRONYMS

ACU	Avionics Control Unit
ACUC	Avionics Control Unit Complex
AMUX	Avionics Multiplex System
BCU	Buffer and Conversion Unit
CITS	Central Integrated Test System
C&D	Controls and Displays
DCR	Design Change Request
DEU	Data Entry Unit
EMUX	Electrical Multiplex System
HOL	High Order Language
IAU	Interface Adapter Unit
IMCT	Intermodule Compatability Test
MTM	Modification Transmittal Memorandum
MVT	Module Verification Test
M&TC	Mission and Traffic Control
RDT	Radar Data Terminal
SPR	Software Problem Report
SVT	Systems Validation Test
S/W	Software

EVALUATION

The need for producing more reliable, low cost software, as stated in such documents as the Command, Control Information Processing CCIP-85 Study (Information Processing/Data Automation Implications of Air Force Command and Control Requirements In the 1980's) has led to the development of software error prediction models for predicting reliability and error occurrences, as well as investigations into the types and causes of software errors, in order to develop ways of producing more reliable, "error-free" software code. However, current model development and error data analysis has been somewhat hampered by the lack of sufficient software error data from actual software projects that can be used as a basis for software model testing and for data analysis.

This effort was initiated in response to the CCIP-85 Study and this lack of software data, and fits into the goals of RADC TPO No. 5, Software Cost Reduction (formerly RADC TPO No. 11, Software Sciences Technology), in particular the area of Software Quality (Software Data). The report focuses on the acquisition of various software error data items, and the problems encountered in trying to collect and categorize that data, from a large avionics software development project for the Department of Defense. The importance of providing this error data is that this data will be used to support software model development and will also be analyzed for discernible patterns in the types and categories of errors as functions of such characteristics as software type and development phase. In addition, the problems in collecting this data, as encountered during this effort, will lead to improved methods for collecting data from future projects to support software error data analysis.

By using this data to develop and test software error prediction models and by carefully analyzing the data, we can determine the nature of software errors and develop tools for accurately predicting these errors. This, in turn, will lead to the production of more reliable software. Finally, the data provided under this effort will be used to help establish a software baseline for avionics software projects in terms of such quantities as types and number of errors, which eventually will lead to development of methods for better controlling future avionics software development projects.

Alan N. Sukert

ALAN N. SUKERT, Captain, USAF
Project Engineer

1 SUMMARY

This report covers activities of Boeing Aerospace Company to provide data to a software data repository being developed by the Information Sciences Division of Rome Air Development Center. The repository is a source of information on the software development process which can be used to support studies of software reliability models, cost models, productivity and maintainability models, and development of a status and reporting system.

The data described in this report was developed by categorizing 2036 Software Problem Reports of errors encountered during development of one phase of a large DOD system. Twenty predefined major categories were used in seven functional areas. Additional data is included about the source of the errors, the type of the correction made, and the time to find and fix the error, all of which was derived from project records. The problem reports were written in the time period from the beginning of configuration management (start of integration testing, approximately) to delivery of the software to the Air Force.

This report is written in six sections. The first is this summary. The second contains an introduction and a description of the scope of the work. In section three is a description of the hardware and software system, the software development process, and the type of data available. Section four contains a description of the method of categorization and the derivation of the other required data items. This section includes the interpretation of the categories and some comments on the difficulties and successes of the various tasks. Section five is a description of the results and section six presents the conclusions.

In general, Boeing and TRW (1) match well in categorization results. There is, in most cases, a close correlation of the data, with poorly correlated results in only a few categories. Also, within the Boeing data, there is a close correlation of categorization distribution within seven functional areas. That is, although the software was built to fulfill widely differing functional requirements, the percentage of each type of error agrees well in most categories. The percentage of design errors is lower, however, than some other studies of software errors. Finally, a separate count of update errors, i.e., errors arising during correction or updating of code, shows these to be more than a trivial percentage of the total errors.

The value of such data is recognized by many software managers as a source of information for successfully planning future projects. The surprising result is not that one can see differences between industry data and among functional areas but that one can get such a high degree of correlation.

2 INTRODUCTION

2.1 Purpose of the Contract

The purpose of this contract was to obtain software error data from a large DOD systems development project for a software data repository being developed by the Rome Air Development Center.

Boeing has supported this objective by providing such data, in the hope that a repository will facilitate research into the software development process. The goal of such research is to obtain insight into the factors which contribute to software reliability; the areas of the development process in which errors arise most frequently; and the impact of such factors on the scheduling and cost of the project.

2.2 Scope

Data was gathered from a Boeing Aerospace Company project for a large software system. The number of error reports provided in the data base is 2036. The software consisted of approximately 80,000 assembly language instructions and approximately 40,000 lines of JOVIAL/J3B instructions (roughly equivalent to 240,000 assembly instructions). This project included operational software and the simulation software necessary to develop and test the former. The categorized software problem reports were written against the first two released blocks of software, Block 0 and Block 1. Block 1 was considered an updated and corrected version of Block 0.

According to the contract "The contractor agrees that all documents produced in the performance of this contract shall include the necessary safeguards for protecting the source of all data for this effort, including the names of the project and all component modules, notwithstanding any other provision of this contract. It is understood that the contractor shall not be required to provide any information or data hereunder which may jeopardize the protection of such data sources."

3 PROJECT INFORMATION

3.1 System Description

The system consists of a controls and displays subsystem, a hardware test monitor, two system functions, A and B, and an executive system which schedules the former functions. In addition, resident on two other computers are a system simulator and a subsystem simulator to provide a test environment.

The overall system is shown in Figure 1. A central portion of the hardware system is the Avionics Control Unit Complex, consisting of:

2 Avionics Control Units (ACUs) (SKC 2070)
1 Mass Storage Unit (drum) - MSU
1 Data Entry Unit (tape cassette) - DEU
2 Interface Adapter Units - IAU
2 Radar Data Terminals - RDT
1 Buffer and Conversion Unit - BCU
Avionics Multiplex System - AMUX

The two Avionics Control Units interface with the other subsystems through the Avionics Multiplex System. Data transmission is two-way, non-simultaneous between the multiplexed terminals, originating with and under the control of one of the Avionics Control Units.

The Mass Storage Unit provides bulk storage for 8 megabits of program and data base information to be passed to and from the ACU memories.

The Data Entry Unit (DEU) provides remote storage on magnetic tape for programs. The tapes on the DEU are removable cartridges.

The Interface Adapter Unit acts as an interface between the Radar Altimeter, the Doppler Radar, the Electrical Multiplex system (EMUX) and the ACUs. It requests data from the radar and the altimeter, stores it and passes it to the ACU when queried. It also contains the EMUX status report to allow the ACU to update EMUX control.

The Radar Data Terminal adapts the Forward Looking Radar and Terrain Following Radar to the system by performing signal conditioning and data buffering.

The Buffer & Control Unit monitors signals from the Mission and Traffic Control System (M&TC) and the Doppler Radar. It digitizes the data for transmission to the Avionics Control Unit Complex for operational status determination of the line replaceable units.

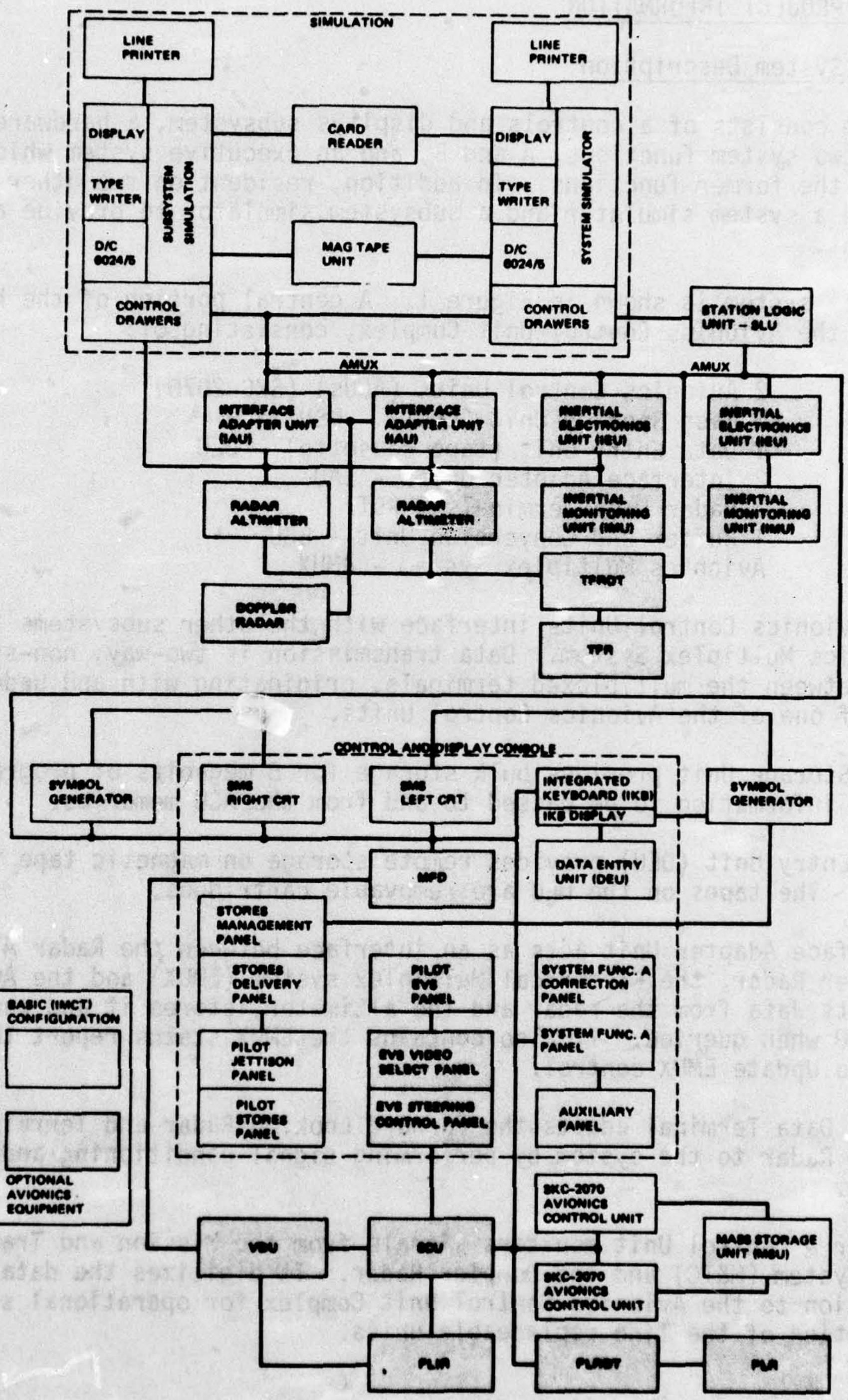

Figure 1. Avionics System

The M&TC system consists of the following communications hardware:

- o UHF/ADF radio
- o HF radio
- o Secure voice
- o IFF
- o Secure IFF
- o TACAN
- o UHF radio
- o X-Band Rendezvous Beacon

Finally, the following comprise the rest of the system:

- o Controls and Displays Subsystem
- o Stores Management System
- o Air Vehicle Electronics
- o Central Integrated Test System (CITS)

The CITS has its own ACU which communicates with the two central ACUs. The CITS performs tests of both avionics and non-avionics subsystems.

3.2. Software Design

The software is designed to consist of five major functional areas in the operational software and two functional areas in the simulation software (Figure 2). Modules exist within functional areas, not across their bounds, and consist of a set of programs with similar attributes.

These functions have been further broken down into basic and non-basic capability. The software is designed so that if one ACU should break down, the system can still provide the basic functional capabilities. These basic capabilities are defined separately for each functional area and consist of a subset of the software. The basic set of all the software to support these capabilities is resident on each ACU. The non-basic set is divided between the two ACUs.

The simulator software runs on two separate computers. The simulator software allows testing to take place in the laboratory as if the system included a real airplane under actual flight conditions and simulates certain other equipment which could not exist in the testing environment.

The operational software operates in two airborne computers using a cyclic algorithm. It operates off a 15.625 ms interrupt, an interval which defines a minor frame. Four minor frames constitute a major frame of 62.5 ms. There are three types of programs:

(a) cyclic - active every major frame

(b) non-cyclic - active in a major frame only on demand

(c) background - active every minor frame when time permits and whose complete execution can be spread over several major frames (interruptible programs).

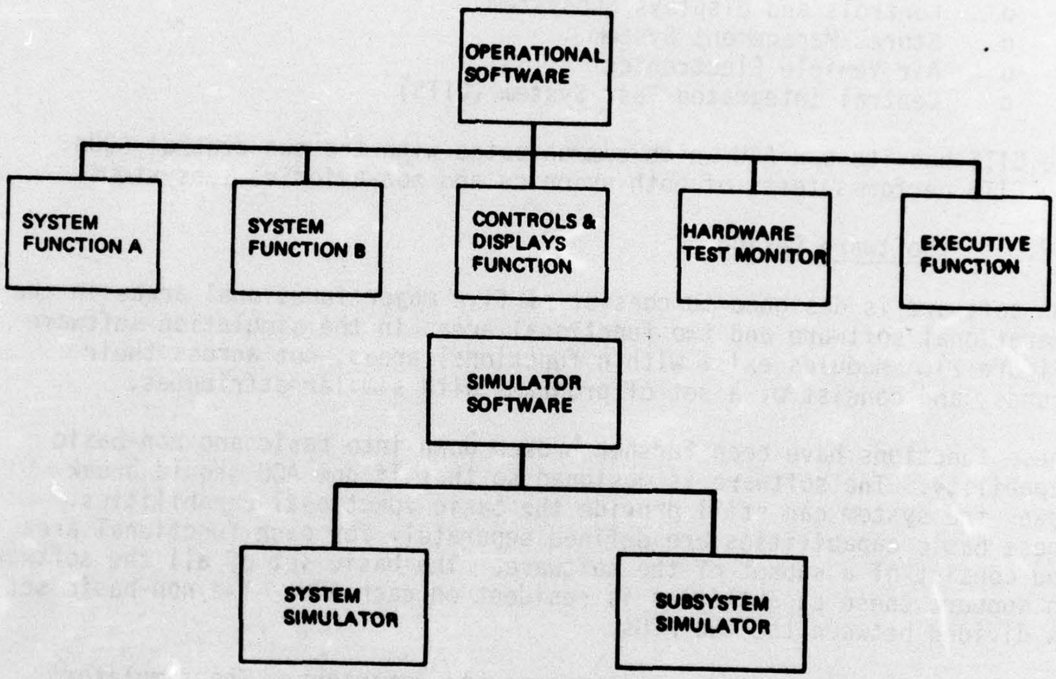

Figure 2. Software Organization

This cyclic operation is shown in Figure 3 along with the operation under backup mode, i.e., when one ACU is down.

3.3 Software Development Process

3.3.1 An Overview

Software for this system was developed using both an IBM 360 computer and development laboratory facilities containing the four computers previously discussed.

All program compilation, assembly and creation of tapes is done using the IBM 360 and a highly developed support software system. The support software package includes a cross compiler which produces code for the airborne computers. It also includes assemblers for both the airborne and simulator computers and various other special purpose support tools, such as linkers, loaders, data base management tools and simulator programs. It is the heart of the software management system.

Early in the software development process, when neither the simulator software nor the operational software was complete, module verification testing was also done on the IBM 360. This testing was done using an emulation program for the airborne computers, since these computers were not yet delivered. However, as soon as a software package with the basic functional capability was complete and the development laboratory set up (by the start of Block 0 Intermodule Compatibility Testing (IMCT)), module verification testing was done in the laboratory using the airborne computers.

3.3.2 Tools and Coding Restraints

The Air Force specified no formal coding standards under which this software was developed. There were three debug tools which were used during program development and two programs which were used to accomplish core and time optimization. Two debug tools were vendor supplied, one was developed in-house by one of the software designers.

Two of the debug tools accomplished data flow analysis. The first, designed into deliverable software, enabled the programmer to display any specified core location on the airborne computers. The display was updated every second so that changes over time could be monitored.

It was also possible to step through adjacent locations and track any changes in these locations in the same one second cycle. This debug process was real-time tracking in the sense that decisions concerning what locations to track were made in real-time during a run.

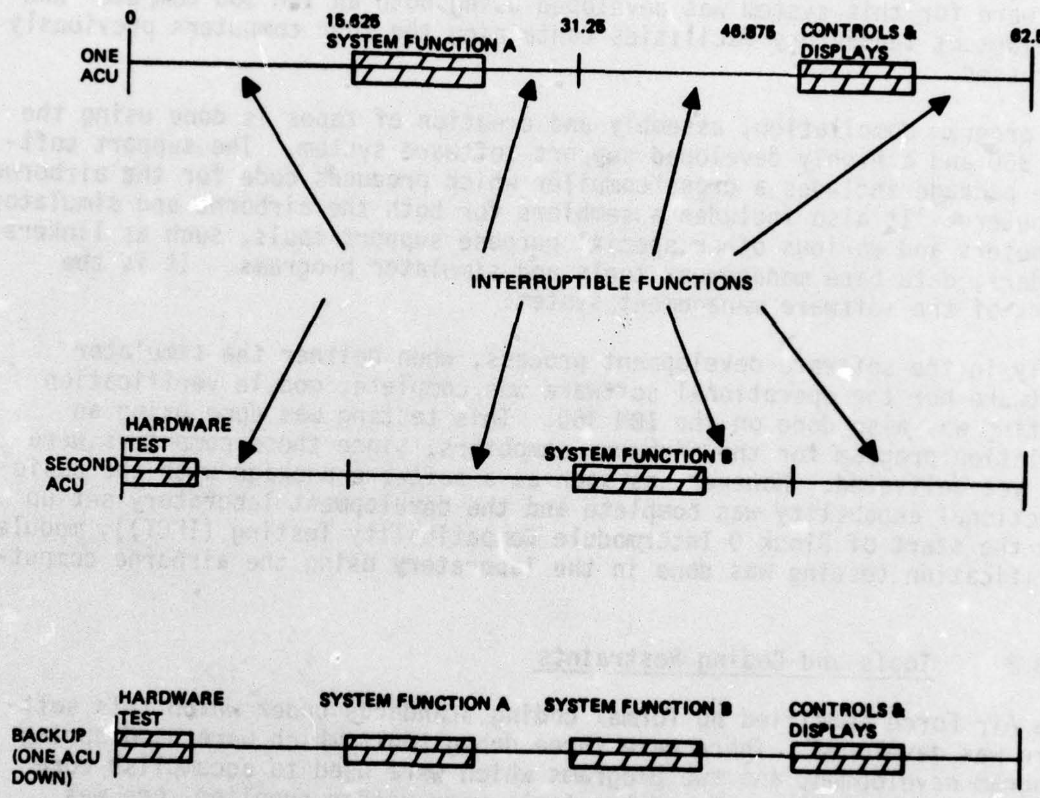

Figure 3. Allocation of Software Function by Time

A second debug program which performed data flow analysis was written by a software designer. Before a test run, a programmer would specify which data in the data base was to be monitored and at what intervals. At these intervals, all the named data items were printed out on the printer attached to the simulator computers. This allowed tracking of changes in the data base items.

Both these tools were heavily used by the project during all phases of the software development process for debugging. The first tool was used more heavily than the second.

A third debug tool, again vendor supplied and used very early in the development cycle, interfaced with specific core locations. A location could be interrogated and the absolute code changed by keyboard input. This did not, of course, operate in real-time. Execution stopped when a key location was reached, at which time the interrogation and input occurred. Initially, this was the only method used to correct programs except for source code update and recompilation. Subsequently, when tapes could be patched, this technique was no longer used.

There was an executive program which was used to identify candidate code for time optimization. Since this was a real-time system, there were timing restraints which required that code be optimized. A job scheduler program was used to compute time spent in different areas of code, based on interrupts from a real-time clock. This program was used to insure timing requirements were met and to identify candidate code for optimization to meet requirements. This work of course was begun fairly early in the development process and extended into the integration testing phase.

A fifth program, written in FORTRAN, was used to aid in core use optimization. There was a contract requirement that at least 75% of the code be written in HOL, the rest in assembly language. This requirement was that the assembly language instructions generated by the J3B compiler be 75% of the total assembly language instructions. The total was the sum of the number of assembly language instructions generated by the compiler and the number of assembly language instructions written directly by the programmers. The core use tracking program kept track of how much core was used by instructions generated from the HOL, how much was required for direct assembly language, and how much core was consumed by data. It was run often, especially after a release of a new program version, to insure contract compliance and to optimize use of core to meet these requirements.

3.3.3 Testing Process

There were three separate testing periods for the software. The first was module verification testing (MVT), followed by inter-module compatibility testing (IMCT), followed by systems validation testing (SVT). Both MVT and IMCT were done within the software organization; SVT was done in the systems test group.

Module verification testing was carried out by the programmer who built the module. This was done informally, i.e., no configuration management requirements were enforced, hence no record of errors found in the module under test were kept.

Very early during initial coding and MVT, this testing was done on an IBM 360 using some emulation software. However, when the operational software was sufficiently complete to allow testing in the development lab, MVT was done using the simulator and airborne computers, with the rest of the software as a testing environment.

The IMCT was carried out by a separate testing group, not part of the software design group. Its job was to develop test plans using the software functional requirements. By starting with a design document which laid out these requirements, tests were developed to check whether the software met them. At this point, the software subsystem existed as a whole and these tests checked the compatibility of the functionally separate pieces of software.

The SVT was carried out by yet another separate group, the Systems Test Organization, which was responsible for checking compliance with system requirements. These requirements described the operation of the whole system including all the hardware (computers and avionics equipment) and the operating software. The tests did not specifically test the software.

3.3.4 Anatomy of an Error

If an error was discovered during MVT, as mentioned above, there was no Software Problem Report issued against the module under test. This was because as far as configuration management was concerned, the software was not released. If during MVT, testing turned up previously undiscovered errors in software already released (which showed up as a result of testing the new module in the total environment), an SPR was issued by the software designer against the already released software, but never against the new module.

IMCT as viewed within the company is an internal acceptance test of the software. It tests the software package as a total unit, checking it against functional requirements. When all requirements are met, the software is released outside the software organization. The IMCT process catches many errors which otherwise would be observed further down the line during systems test or flight test when errors are possibly more expensive to fix.

During IMCT, if an error was discovered, the test engineer wrote up a description of the problem. Based on the functional requirements the person was testing and based on the person's knowledge of the software, the test engineer contacted the appropriate software designer. If a fix was necessary to allow testing to proceed, the fix was done by patching the program tape. The source code version (on another tape) was also

corrected. At appropriate times during IMCT this corrected source program was recompiled to form a new master tape and used for the testing. If the error could not be conveniently patched and was not critical to the current testing, the only fix effected might be correcting the source code for the next compilation.

After IMCT, the tape was released from the software development group to the SVT group. The purpose of SVT was an acceptance test of the system for Quality Control. The handling of the software during SVT differed from IMCT in two ways. First, the policy was to make fixes by patching the tape, i.e., the tape was the final product. No recompilation of source code to produce new tapes was made unless the fix couldn't be done by patching. A side effect of this was that the final sign off of SPRs opened during SVT did not occur until the next tape was released (after SVT), since the new source code being updated and tested was regarded as the final fix.

Second, the method of testing during SVT was quite different from IMCT. During SVT, many "dry runs" were made to isolate and correct errors. When all the errors were eliminated, a final run for the benefit of Quality Control was made. This resulted in a sign off of the tape which was then released for flight test.

On the other hand, during IMCT, which was internal to the software development group, testing was incremental, i.e., errors were corrected as found, and then testing proceeded.

3.3.5 Software Correction Procedures

When an error was discovered during testing, the usual procedure was to patch the program.

Patching was done on programs resident in core. If the changed code was no larger than the original code, the absolute code in the appropriate core locations was changed to the corrected code. If the changed code was larger than the original, a scratch area of memory was used to create a section of new corrected code with branch instructions used to go around the incorrect code. Roll-in programs were not patched as above since it was too difficult to patch on the fly after they were rolled in. Correction had to be accomplished by updating source code and recompiling, in these cases.

In addition, a program was developed for patching the tapes themselves. It ran on the simulator computer and took two tapes as input, one containing the patches and one the incorrect programs, producing a corrected tape as output. This was the most popular way of patching and simplified the correction process considerably.

3.3.6 Production Control Procedures

In order to assure control over the changing and developing software package, several configuration management controls were in effect.

There was a Computer Program Library which was a central repository for all system software products. Its maintenance and use was integrated with the support software system described earlier. This included test materials, milestone documents, program listings, card decks and system files, all test software, simulation software, and support software.

A master version of all programs was kept at all times and new master versions were created as a result of design changes and software errors. This new version would then go through "regression testing", in which a selected group of tests, passed by the previous version, were repeated on the new version.

All changes were tracked with appropriate paperwork so that no undocumented changes to the software were made. Any software errors were documented on Software Problem Reports, while errors in requirements were reported on Design Change Requests. When a programmer wanted to make any changes to a computer program, these changes were submitted by the programmer to the Computer Program Library with a Modification Transmittal Memorandum. These three pieces of paperwork provided the basic control tracking on the software. They are shown in Figures 4, 5, 6, and 7.

3.4 Sources of Data Other Than SPRs

As in most large projects there was no lack of paperwork generated to track and manage the software. However, as is well known to anyone who has tried to collect data for various purposes, this information is seldom available in a form easily adaptable to other uses. For the purposes of this study there was a great deal of data available. Some of it was used directly, while other data formed a starting point for the derivation of required data. The SPR (Software Problem Report) was the basic source of categorization data. This was the official method of reporting and resolving software errors. Two SPRs are shown in Figures 4 and 5. The first form was an earlier version; at a later point the second form became the official one.

There was another form, a DCR (Design Change Request) which was related to software changes, but not to software errors. Any time an error was found in which it was determined that there was an error in the stated requirements of the software, a DCR rather than an SPR was written. In other words, the error was a result of incorrect requirements rather than incorrectly coded software. No DCRs were included since the study was to include only Software Problem Reports. DCR's were not written against the code, but against a design document.

A summary record of all SPRs was kept, and updated computer listings of these records were available. These did not include duplicate reports or non-software errors.

SOFTWARE PROBLEM REPORT

SPR No. ____________

PROBLEM: (Prepared by User)

Originator ____________ (Name) ____________ (Organization) ____________ Phone No. ____________

System, Processor, or Component Failing or Project Involved ____________ Computer ____________ System Version ID ____________ Test Case or Program ID ____________

Classification

- ☐ Minor or Not to Specs.
- ☐ Major or Missing
- ☐ Information
- ☐ Revision Request
- ☐ Software Addition

Description of Problem (Attach additional pages if necessary -- include line numbers or other identification of offended statements or data)

____

Correction Required By ____________ Date

Enclosures

- ☐ Program Listings
- ☐ Run Deck
- ☐ Run Instructions
- ☐ Storage Map Listings
- ☐ Data Listings
- ☐ On-Line Output

Authorizing Signature ____________ (Name) ____________ (Organization) Date ____________ Time ____________

ANALYSIS: (Prepared by organization responsible for software)

Received Date ____________ Time ____________ Charge Number ____________

- ☐ Software in Error
- ☐ Software Not in Error Explain and Return to Originator
- ☐ Insufficient Information for Analysis. See Explanation
- ☐ Error Previously Reported On SPR No.
- ☐ Others, Explain
- ☐ Not Approved
- ☐ Approved for Correction or Change

Explanation: ____________

Analysis Time Expended:

Man Hours ____________

Computer Hours ____________

Computer ____________

Estimated Cost of Solutions:

Man Hours ____________

Computer Hours ____________

Planned Correction Date ____________

Signature ____________ (Name) ____________ (Organization) Date ____________ Time ____________

JCD-122

CORRECTION: (Brief description of work performed, including test cases used to confirm correction)

Solution: ____________

Modules Changed ____________

Correction Time Expended:

Man Hours ____________

Computer Hours ____________

Submitted to ____________

Work Performed by (Signature) ____________ Date ____________ Time ____________

CONFIRMATION: Corrections Verified by Product Assurance ____________ (Signature) Date ____________ Time ____________

MTM No(s) ____________

Available in (Version ID) ____________ Date Returned to Originator ____________ Time ____________

R-26400 ORIG. 2/72

White = Originator Open | Green = Analysis | Canary = Orignator Closed | Pink = Product Control Closed | Gold = Product Control Open

Figure 4. SPR Form — Version 1

SOFTWARE PROBLEM REPORT

SPR No. ______

PROBLEM: (Prepared by User)

Originator's Name ______ Organization ______ Phone No. ______

System, Processor, or Component Failing or Project Involved ______ Computer ______ System Version ID ______ Test case or Program ID ______

Description of Problem: ______

Classification

☐ Error

☐ Information

☐ Revision Request

Correction Required by Date ______ Reference UER No. ______

Authorizing Signature ______ Organization ______ Date ______

ANALYSIS: (Prepared by organization responsible for software)

Received Date ______ Time ______

☐ Coding Error

☐ Design Error

☐ Software Not in Error, Explain

☐ Error Previously Reported On SPR No. ______

☐ Others, Explain

Explanation: ______

Documentation Impact Milestone

1 ☐ 2 ☐ 3 ☐ 4 ☐

5 ☐ 6 ☐ 7 ☐ 8 ☐

Signature ______ Organization ______ Date ______

CORRECTION: (Brief description of work performed, including test cases used to confirm correction)

Solution: ______

Mod/Programs Changed ______ ______ ______ ______ Hand Load ☐ Yes ☐ No

Work Performed by (Signature) ______ Date ______

CONFIRMATION: Corrections Verified by Product Assurance

Signature ______ Date ______

MTM No.(s) ______

Available I. (Version ID) ______

X 29400 REV 1/75 J40-122

WHITE = Originator Open GREEN = Analysis CANARY = Originator Closed PINK = Product Control Closed GOLD = Product Control Open

Figure 5. SPR Form- Version 2

D E S I G N   C H A N G E   R E Q U E S T
(SOFTWARE)

DCR No. ________

Originator ______________ (Name) ______________ (Organization) ______________ Phone ________ Date ________

Change Title: ______________________________

Change Category

Affected Programs (List all module and program units)

Performance Improvement — MAJOR ☐ ____________ MINOR ☐ ____________

Interface Efficiency Improvement — MAJOR ☐ ____________ MINOR ☐ ____________

Detailed Description of Change (Attach additional information as required)

Cost Estimates:

Programmer ______________ MHS

Documentation ______________ MHS

Machine Time ______________ HRS

Time to Complete ______________ Weeks

Other ______________

Software Change Board Action:

☐ REJECTED Reason: ______________

☐ ACCEPTED Implementation Schedule: ______________

Planned Version for Inclusion ______________

Development Approval

Signature — Group Engineer ______________ Date ________

Product Control Authorization

______________ (Signature) ________ (Date)

Design Change Approval

______________ Software Design Manager ________ Date

Figure 6. DCR Form

MODIFICATION TRANSMITTAL MEMORANDUM

☐ Program Element or ☐ Document or ☐ Test Case	Base Version ______	**For Product Control Use Only**
	Element Name: ______	MTM No.: ______
ATTACHMENTS:	SPRs Resolved: ______	Date: ______
		New Version: ______

☐ New Source

☐ Compiler/Assembly Listing

☐ Assembly Listing

☐ Test Output

☐ Test Case(s) (List) ______

☐ New Test(s)(Include Card Decks, Listings, and Test Output)

Description of Changes for Version Description Document:

______

Description of Remaining Scaffolding or Debug Points:

______

Dependencies (Specify any actions/modules which may be needed prior to integration and/or testing):

______

Documentation Changes (Attach edited pages for Internal Manual/Specify changes required in manuals and specs):

______

List Test Points Satisfied by this Change:

______

Submitted By: ______ Date: ______

Approved By: ______ Development Date: ______

Approved By: ______ Product Assurance Date: ______

J40-006 R-20403 ORIG. 2/73

Figure 7. MTM Form

4 DATA ACQUISITION

4.1 Type and Evaluation of Data Acquired

4.1.1 Categorization

All the necessary information to do categorization was contained on the SPR form. Besides descriptions of the problem as seen by the testing engineer, the form contained the explanation and correction done by the software designer. These latter descriptions were often detailed down to the affected statement level. Space was provided for the testing engineer to indicate whether the purpose of the report was to record an error, relay information or request a revision. That feature proved to be very helpful to this analyst in making determinations from sketchy information. When this help was lacking, it was often difficult to tell whether the software was not working per requirements or the testing personnel were registering a complaint/request for a change. Occasionally a testing engineer would check both the error and revision request boxes, which was interpreted to mean "Here's an error, fix it!"

The software designer had the opportunity to indicate whether the software was really at fault or whether it was a hardware error, operator error, etc. Duplicate reports were also found and indicated on the SPR. These were frequent. They often occurred because a test engineer would write descriptions of several system level problems which turned out to be traceable to one software error.

4.1.2 Module Information

On the SPR form was also noted those programs/modules changed as a result of correcting the error. As explained in Section 3.2, modules are a higher level organization than programs. Sometimes the specific programs were designated but this information was not consistently available. An additional check of this information was available in computer listings of summary SPR information, which was kept independently of the actual SPR reports. This also notes the modules affected by the error.

It would have permitted more meaningful analysis if the programs affected were identified in every case. This is true for several reasons. One, a program is the smallest compilable unit whereas modules are frequently large and rather weakly tied functionally; second, some modules are combinations of programs written in both HOL and assembly language. This alone makes it difficult to evaluate the error rates using HOL vs. error rates using assembly language. Third, for that part of the data where there is information on the actual program, there are indications that certain programs had high error rates while other programs had no reported errors after MVT. The fact that the program name is not always given makes any inferences from this type of information unsupportable.

4.1.3 Termination Information

A determination of abnormal/normal termination was made based on the SPR problem description. The decision that there was abnormal termination was based on a description of (1) infinite loop, (2) system crash or (3) reference out of memory bounds. There is no assurance that all cases of this were reported specifically in the problem description, therefore these results are probably weak.

In addition, this type of information was not kept in this form by the project. In fact, it is doubtful that it would have been informative to do so in this environment. In a real-time environment, the situation is more complicated than in a batch environment. Jobs are not aborted by an operating system which is scheduling and maintaining a batch environment. Here the objective is to keep a system running under even non-ideal conditions, only aborting when non-recoverable errors occur. Also, the environment was not one large computer but four, with sensors and display equipment attached. In a sense, all errors were reports of abnormal conditions. These reports included many problems overlapping the interfaces between the software and the system hardware. It would be more enlightening to know where in the system an error manifested itself, i.e., at which interfaces did most symptoms of errors appear.

4.1.4 Development Information

This refers to the designation of the point in the software development cycle where the error occurred, i.e., was it a design or coding error? This determination was also made from the SPR form. In about half of the cases examined, the programmers marked the boxes provided. The rest were either reported on early SPRs where such a box was not available or reported without the inclusion of this information. In such cases, the analyst made the decision based on an evaluation of the description of the problem and its correction, i.e., subjectively, but hopefully in an informed way.

A note here: a third category was added by the analyst, i.e., errors that occurred as a result of an earlier error correction. These were frequently spelled out on the SPR and it seemed important to get some measure of the number of errors which are injected into the system as a result of attempts to fix those present at the start of testing. Reliability models often make the assumption that there are none of these or that the number is trival. This study found as a conservative estimate that 6.5% of the total errors were specifically reported as update errors.

While the value of such information seems obvious, (i.e., by knowing at what point in the software cycle errors occur, we know where to spend our money and effort), this study at best yielded numbers of dubious value. It is suggested that to get data of this type, two things should be done. First, the only person who knows the development phase of the error (design, coding or update) is the designer. Anyone else's guess is just that. Second, this analyst would be very skeptical of any figures

bandied about by people without (1) a clear definition of what constituted each of these types of errors, preferably defined in terms of the documentation of the project (i.e., what documentation constitutes the design, where does design end and coding start) and (2) assurance that the figures were developed using strict definitions understood by everyone developing the data.

4.1.5 Timing Information

This kind of information was the most difficult to collect. CPU time on a large batch operated computer facility is tracked carefully. This is a matter of economics since people are billed according to the amount of resources their job actually consumes, including CPU time. In a real-time system, it is doubtful that CPU time would mean anything even if it were traceable. When the computer system in the lab was being used, all four computers were running.

The lab was part of the project equipment and resources. As such, it was at the disposal of project staff 24 hours a day and scheduling was necessary when conflicting demands were made on the equipment. Because of this, there was no financial data kept on the development lab that would even allow attributing calendar time to specific test runs at any stage in the development cycle. There were records of who worked what shift during IMCT, but only since some test engineers carefully kept many varied types of records.

It should be mentioned here that some development work was and is done on an IBM 360. This includes compilation of all JOVIAL source code, assembly, and generation of load tapes. In this casé, finance data of a general nature is available. However, it is not directly related to time to fix an error, or to elapsed testing time until an error was discovered.

This analyst looked at all the available records kept by the testing groups. This included the testing log. The response of all testing personnel was that such data as requested in the contract was not available. This points strongly to the need to define requirements for software data collection before a software development project begins, when one can influence the form in which records are kept. Records are kept in various forms, but often the aims of many groups could be met by collecting one unified set of data.

The approach taken finally was to derive time until discovery of an error, based in part on records of shifts worked during IMCT and SVT, in part on some assumptions made by the analyst, and in part on the opening date of the SPR.

Testing time until an error was discovered was based on the date the first SPR was written against the specific software functional group. That is, the date the first SPR was written against the software in one function was considered Day 1, start of test for all that software. This was necessary because there was no official date when all software in any functional area was considered complete as a unit and officially released. Thus, it was necessary to approximate this date by using the SPR date. Otherwise, all SPRs prior to start of IMCT would have had to be ignored, and Day 1 would be start of IMCT of Block 0, even though the software was under configuration control prior to this point. Indeed, for three functional areas, much testing and many SPRs had already occurred prior to IMCT.

One cannot assume by the above that the software existed as a whole final unit on this day. Some software was still being built at this point and would enter the testing cycle later. Also, previous to this Day 1, various modules had undergone varying degrees of MVT, depending on allowable time. Considering all the data available, however, this seems the best way to determine the starting date for testing of all software, since it is consistent across all functional areas.

The calculation was made by using accumulated hours of testing during formal test plus an assumed one hour/day/functional area of testing time prior to formal testing, all based on elapsed time since Day 1 to the day the particular SPR was opened. The formula used in this calculation was:

$$\text{Test Time} = \sum_{D\,=\,\text{Beg. of Test}}^{D\,=\,\text{Date Error Discovered}} EH_D$$

EH_D = Number of daily hours of equipment use

Time to fix an error was calculated based on the number of days an SPR was open and an assumed 8 hr/day of equipment use to fix. This 8 hours was divided up among the errors open on any one day, and this fractional time was summed up over the days the SPR was open, to give the final total time spent fixing an error. The formula for time to fix an error was:

$$\text{Fix Time} = \sum_{I_O\,=\,\text{Day of Discovery}}^{I_F\,=\,\text{Day of Closing}} H_{E_I}$$

I = i^{th} day

H_E = $8/SD$ = Hrs Spend Correcting an Error on Day D

SD = Number of Errors Open on Day D

It is this analyst's opinion that CPU time in a real-time environment should not be collected. In fact, the accurate collection of machine time to correct errors would require a mobilization of programmer cooperation that would be difficult if not impossible. If collected, such information should seemingly be augmented by adding in the amount of desk time spent by the programmer debugging and doing mental software testing. Of course this would be equally difficult to collect. Early SPR forms had space for both of the above quantities to be filled in by the programmer, but this part of the form was almost universally ignored by the programmers. The press of project work evidently does not encourage such careful tracking of time.

It seems to this analyst that models which require accurate tracking of CPU time or computer time to do prediction will find little applicability in this environment.

4.1.6 Correction Information

It was necessary to indicate the type of correction and this information was provided in two parts. The first was a description of the correction as code, design or data change. Code and data changes are self explanatory and were easily determined. A determination was made to designate a change as a design change only if a DCR (Design Change Request) was written as a result of the error. A DCR was written when the original requirements were actually in error; the software as written faithfully implementing these faulty requirements. It would have been preferable if all SPRs had designated whether or not a software design document had to be changed as a result of the error, but this information was not provided in all cases. Such changes could then be considered true design changes. Certain documents were software design documents and some problem reports noted when these documents were affected by an error. Of course, even with this information available, picking one of these may have no meaning. An error which forces a design change also commonly causes code changes. And some errors may cause code and data changes, or all three.

The second was an indication of whether or not a correction involved an addition, deletion, correction or all three. Sometimes it was easy to tell if an addition or deletion of code or data had occurred, but it was rarely all three. However, all real software errors required some type of modification to the code. Thus, given no information at all, this was the default choice. For many other errors it was the only proper choice; a true modification of the code had to be made.

4.2 Interpretation of the Categories

Many of the categories were self-explanatory, while many others were subject to interpretation. (See Appendix A). The task of interpretation would have been much easier had a description of the categories been documented. Such documentation, possibly a brief one sentence description of each subcategory, would have made the job of the analyst easier.

It would help assure uniform application among different analysts. Categories which seem obvious to the person who developed them on the basis of observed errors are often obscure to the person using them. In fact, it would seem that documentation, although sometimes apparently superfluous, is a necessary part of the task of developing a tool to be used outside the domain of the developers.

Examination of the final TRW report (1) reveals that many of the evident problems associated with their original categories have been eliminated by their new, shorter list. In fact, the new list seems very usable and superior to the first list. However, for the sake of completeness, some discussion of the twenty categories used in this study is necessary. First, there are too many categories and subcategories in all. This analyst tended to categorize by using a manageable subset of the subcategories which appeared to cover most situations. Only when an error presented classification problems did she sift through the full set looking for a new category which could be applied. In other words, a subset would have been sufficient for the job. Clearly, to be effective, the category list should only be as long as can be comfortably housed in the analyst's mind.

Second, some categories were patent misnomers. The subcategories of I/O Errors describe only output problems. User Interface Errors describes problems with input.

Third, the Recurrent Errors category contains two subcategories, one a recurrent problem, the other a duplicate report. The first subcategory is a real error, the second is not. Thus, if one looks at the whole category when examining the results, one's conclusion could be erroneous. These two subcategories should not be grouped together.

The Logic Errors category contained subcategories which were so general and easy to apply that many errors found their way into it. One subcategory was incorrect logic, another was missing logic. In a general sense, these describe a majority of errors. Another category, Requirements Compliance Errors, included required capability overlooked or not delivered at time of report. This was generally applicable to many error descriptions where the testing group detected some functional capability missing. Decisions for differentiating the latter from missing logic were made on the basis of the detail present in the Analysis or Correction section of the SPR. If the missing capability was traced to some small bit of missing code, Logic Error-missing logic was chosen. If the error report gave only enough detail to determine a missing capability, or if the error resulted in the addition of a large piece of code to supply some missing capability, a Requirements Compliance Error was chosen.

Category E, Operating System/System Support Software, was essentially not used because problem reports written against the support software were not included in this study. They were written as SPRs on a separate

functional area. A few did slip through erroneously written against other functional areas and thus appear in the results. That is, an error at first attributed to one of the seven areas studied here turned out to be an error in the support software. Then the initial SPR was closed and a new SPR written against the support software. There would be no problem reports written against the operating system. The project did not use the operating systems provided with the airborne and simulator computers. Instead, an executive system was written by the software designers for both computers. All Software Problem Reports written against this executive software were included in the study. Rather than putting all of the approximately two hundred in one subcategory they were categorized according to the cause of each separate problem, i.e., the executive system was just one more functionally separate component of the whole project software package.

Except for compilation errors, the Configuration Errors category was seldom used. Compilation errors in general were not reported since software designers would not consider these a type of error to report. Instead they would merely correct the source of the compilation error without issuing any Software Problem Report.

It was hard to deal with the categories Data Handling Errors and Preset Data Base Errors, subcategories MM030 and MM040. It was not always possible to distinguish an item in the data base from a local variable. The analyst used the description on the SPR and tried to make the best decision based on the sense of the discussion. If it seemed to be a locally used variable, the DD (Data Handling) category was used; if it appeared to be in the data base, the MM category was picked. The choice was occasionally arbitrary.

Some Documentation Errors seemed to be hard to establish. For example, operator errors in which the testing engineer mistakenly reported a software error where there was none could be caused by a misinterpretation of the requirements on the part of the testing person or by an error in the documentation of the testing requirements. It was not easy to differentiate the two.

5 RESULTS

This section contains the more interesting results of this study. It surveys the data from different viewpoints but is not meant to be exhaustive.

5.1 Categorization Results

A summary of the results of the software error data categorization work is contained in Table I. A histogram, which gives a pictorial representation of the categorization results, is shown in Figure 8. A breakdown of categories into their major subcategories is presented in Table II. Figures 9A, 9B and 9C show the categorization results in the seven functional areas studied. Figure 10 is a comparison of the Boeing and TRW data (1).

By percentage, the top 7 sources of errors are as follows:

Logic
Data Handling
User Requested Changes
Operator
Recurrent
Requirements Compliance
Computational

These error categories are all those comprising more than 5% of the total errors.

By far the largest percentage of errors are Logic errors, almost one-third of the total. The second largest percentage is Data Handling errors at 13.4% When all data related category errors are totalled, i.e., Data Handling, Data Base Interface, Preset Data Base and Global Variable/Compool Definition, the result is 19.8% or one-fifth of the total. All interface errors total 3.9%, a rather low percentage. The percentage of Computation errors is non-trivial at 5.4%. A comparison of these results with those of TRW represented in Table III shows that the first two, Logic and Data Handling, match well with the results of several of the projects studied by TRW. The other high ranking results, with the exception of the Computational and User Requested Changes categories, do not occur on the TRW lists.

A statistical measure of the difference between the Boeing and TRW results was computed by a Chi-square test applied to the two sets of data, using the average of the two sets as the population value. The Chi-square test can be used to test the hypothesis that the distribution of results by category are not different between Boeing and TRW. Observed differences may be due to chance - i.e., sampling error. An α of .05 was used, where α is the probability of rejecting the null hypothesis when it is true.

Table I. Summary of Categorization

CATEGORY		NUMBER	PERCENT %
A	COMPUTATION	109	5.4
B	LOGIC	636	31.2
C	I/O	28	1.4
D	DATA HANDLING	272	13.4
E	OS/SYS. SUP. S/W	8	0.4
F	CONFIGURATION	12	0.6
G	ROUTINE/ROUTINE INTERFACE	41	2.0
H	ROUTINE/SYS. S/W INTERFACE	3	0.2
I	TAPE PROCESSING INTERFACE	5	0.3
J	USER INTERFACE	12	0.6
K	DATABASE INTERFACE	17	0.8
L	USER REQUESTED CHANGES	161	7.9
M	PRESET DATA BASE	67	3.3
N	GLOBAL VARIABLE/COMPOOL DEF	46	2.3
P	RECURRENT	148	7.3
Q	DOCUMENTATION	27	1.3
R	REQUIREMENTS COMPLIANCE	144	7.1
S	UNIDENTIFIED	30	1.5
T	OPERATOR	158	7.8
U	QUESTIONS	19	0.9
V	HARDWARE	32	1.6
X	NON-REPRODUCIBLE	62	3.1

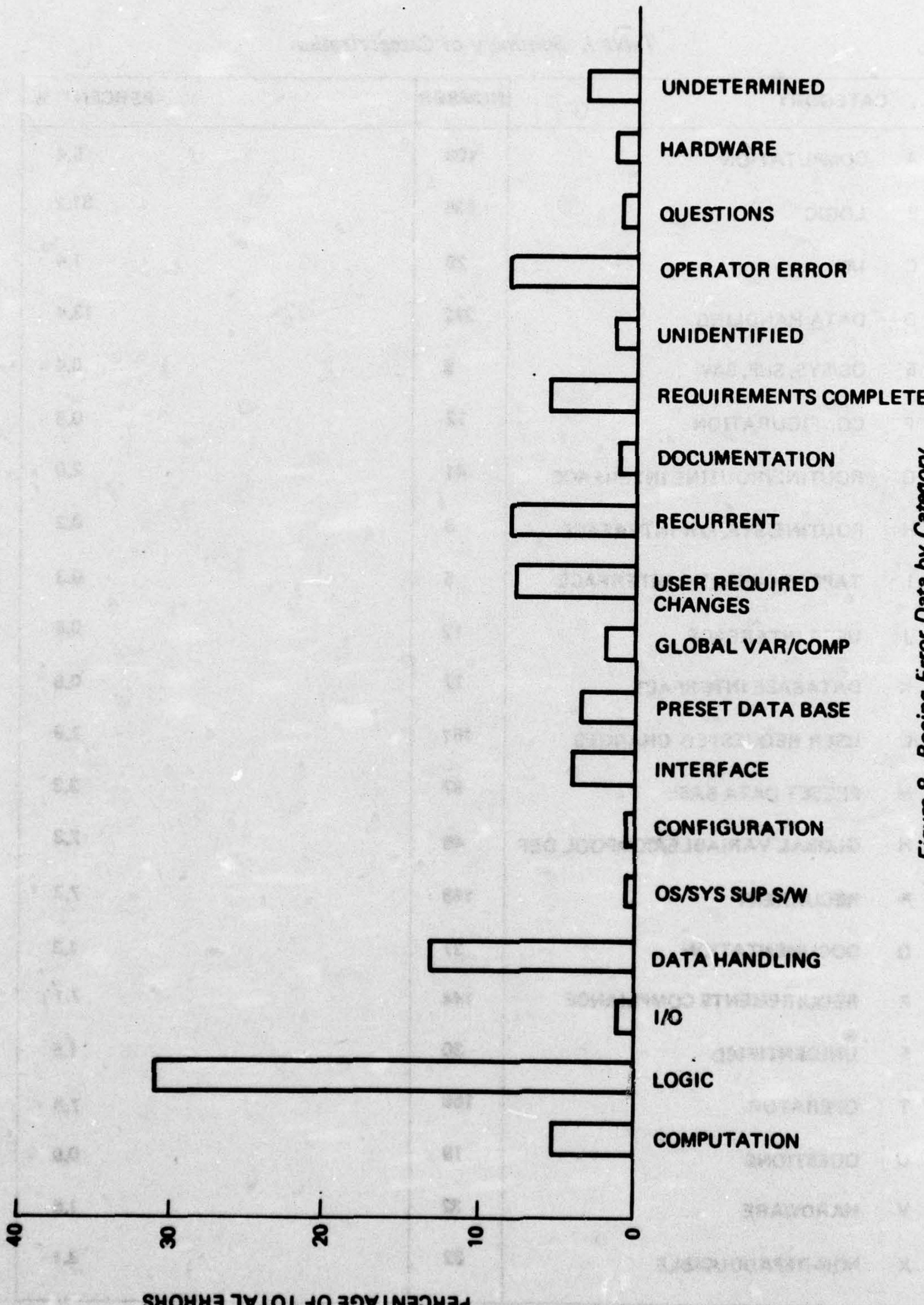

Figure 8. Boeing Error Data by Category

Table II. Major Subcategory Results

CATEGORY SUBCATEGORY	CATEGORY PERCENTAGE	SUBCATEGORY PERCENTAGE
COMPUTATION	5.4%	
WRONG EQUATION/ MATHEMATICAL MODELING PROBLEM		56%
SIGN CONVENTION ERROR		12%
LOGIC	31.2%	
MISSING LOGIC OR CONDITION TEST		29%
INCORRECT LOGIC		34%
PHYSICAL CHARACTERISTICS OF PROBLEM TO BE SOLVED, OVER-LOOKED OR MISUNDERSTOOD		19%
I/O	1.4%	
MISSING OUTPUT		36%
OUTPUT FORMAT ERROR		25%
DATA HANDLING	13.4%	
DATA, INDEX OR FLAG NOT SET OR SET/INITIALIZED IN-CORRECTLY		41%
DATA, INDEX OR FLAG MODIFIED OR UPDATED INCORRECTLY		34%
USER REQUESTED CHANGES	7.9%	
NEW AND/OR ENHANCED FUNCTIONS		48%
DATA BASE MANAGEMENT AND INTEGRITY		14%
EXTERNAL PROGRAM INTERFACE		17%
PRESET DATA BASE ERRORS	3.3%	
NOMINAL, DEFAULT, LEGAL, MAX/MIN VALUES		39%
PHYSICAL CONSTANTS AND MODELING PARAMETERS		34%
GLOBAL VARIABLE/COMPOOL DEFINITION	2.3%	
DATA DEFINITION		43%
LENGTH OF DEFINITION INCORRECT		26%
DELETE UNNEEDED DEFINITIONS		22%
RECURRENT ERRORS	7.3%	
PROBLEM REPORT REOPENED		22%
PROBLEM REPORT A DUPLICATE OF PREVIOUS REPORT		78%
REQUIREMENTS COMPLIANCE	7.1%	
REQUIRED CAPABILITY OVERLOOKED OR NOT DELIVERED AT REPORT TIME		82%
OPERATOR ERRORS	7.8%	
TEST EXECUTION ERROR		87%

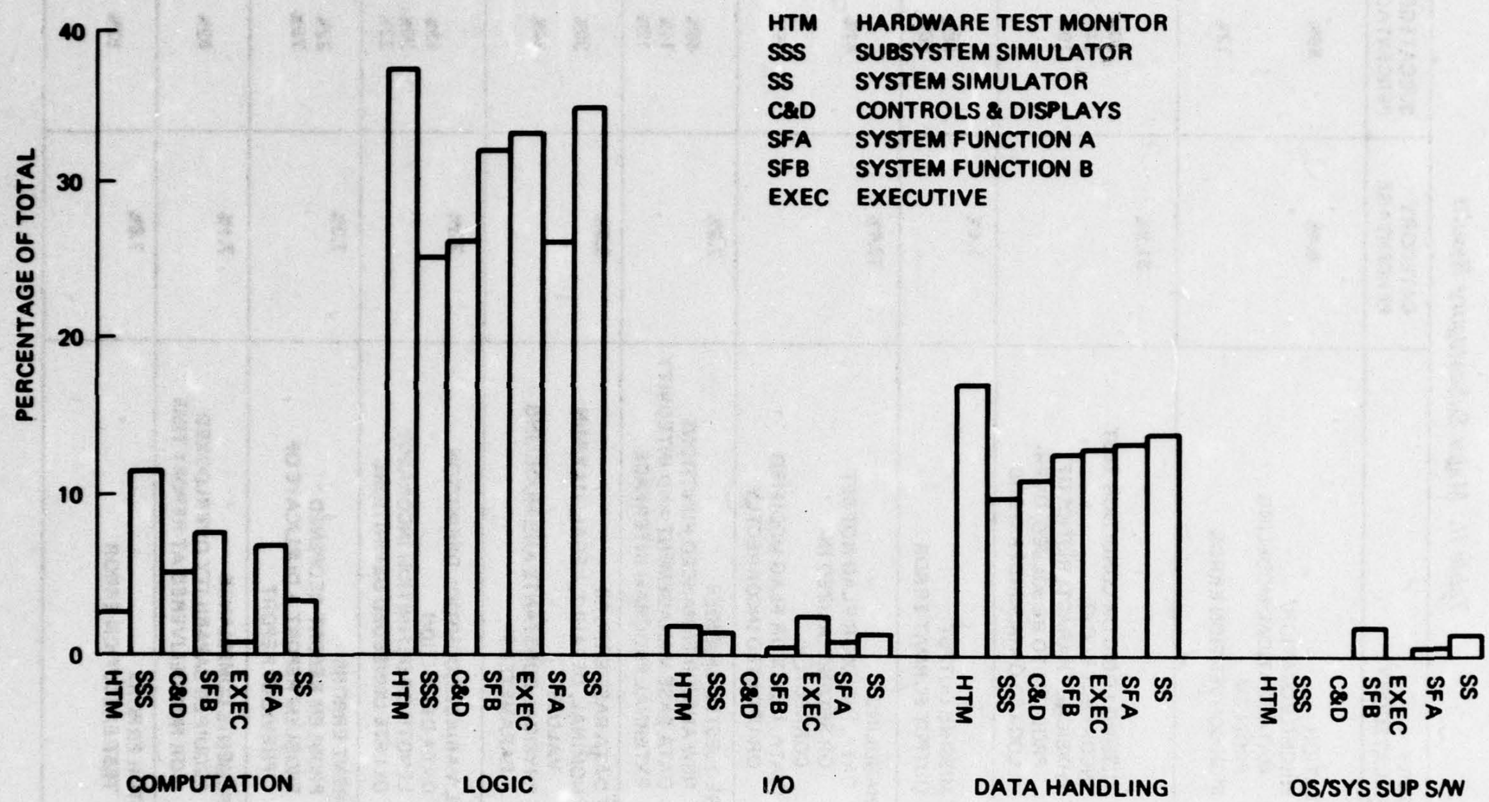

Figure 9A. Boeing Error Data By Function For Categories A Through E

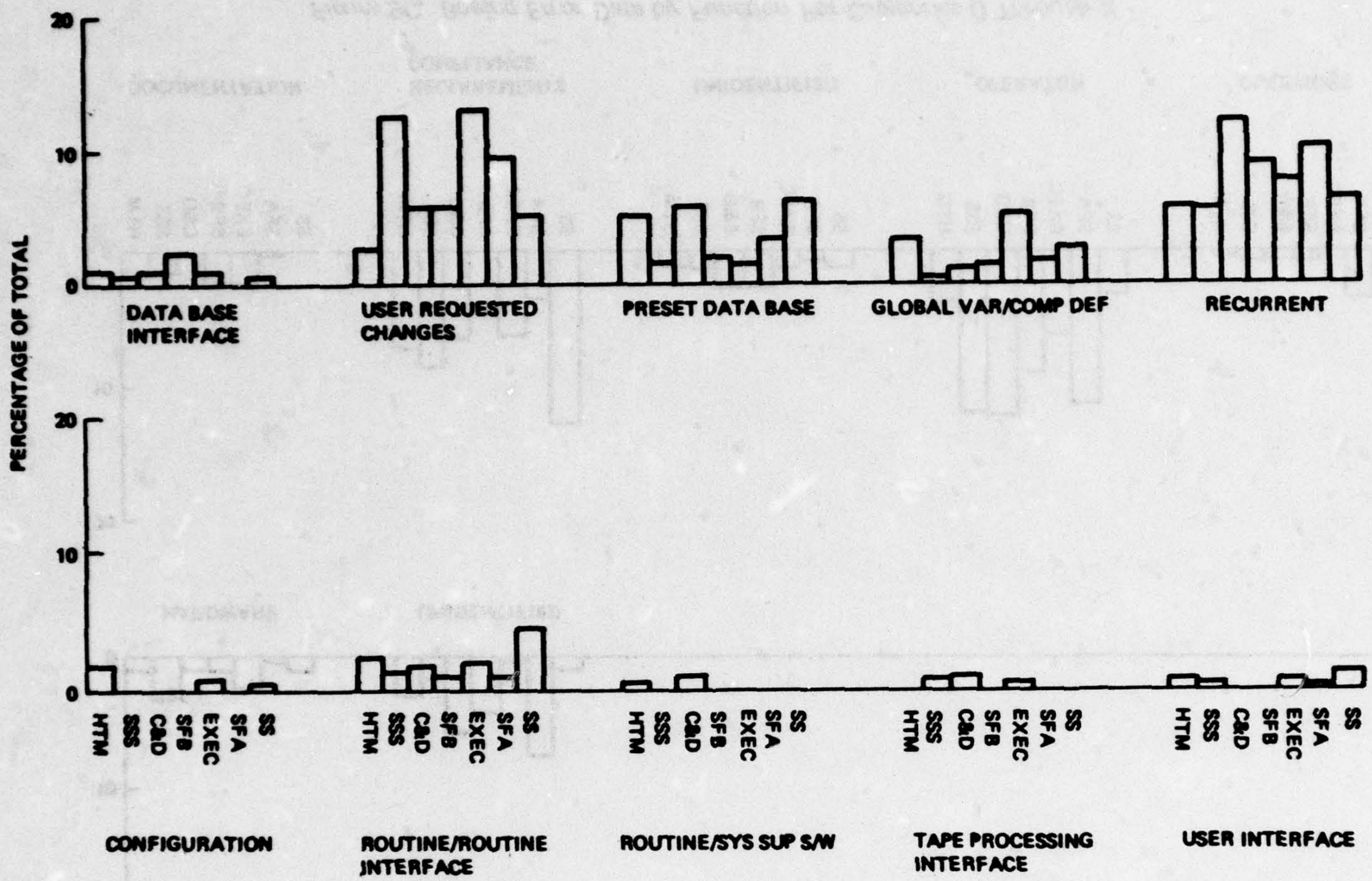

Figure 9B. Boeing Error Data by Function For Categories F Through P

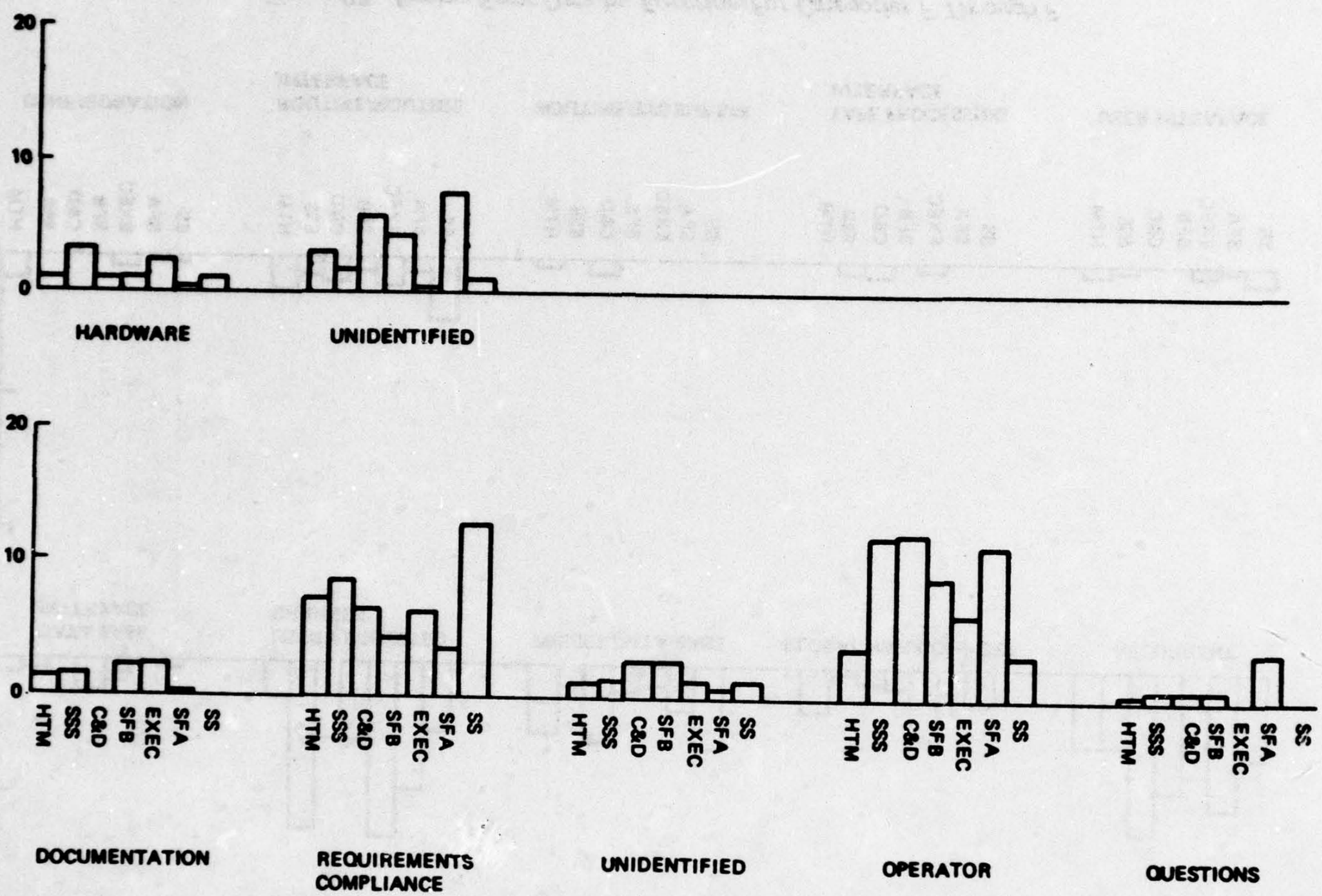

Figure 9C. Boeing Error Data by Function For Categories Q Through X

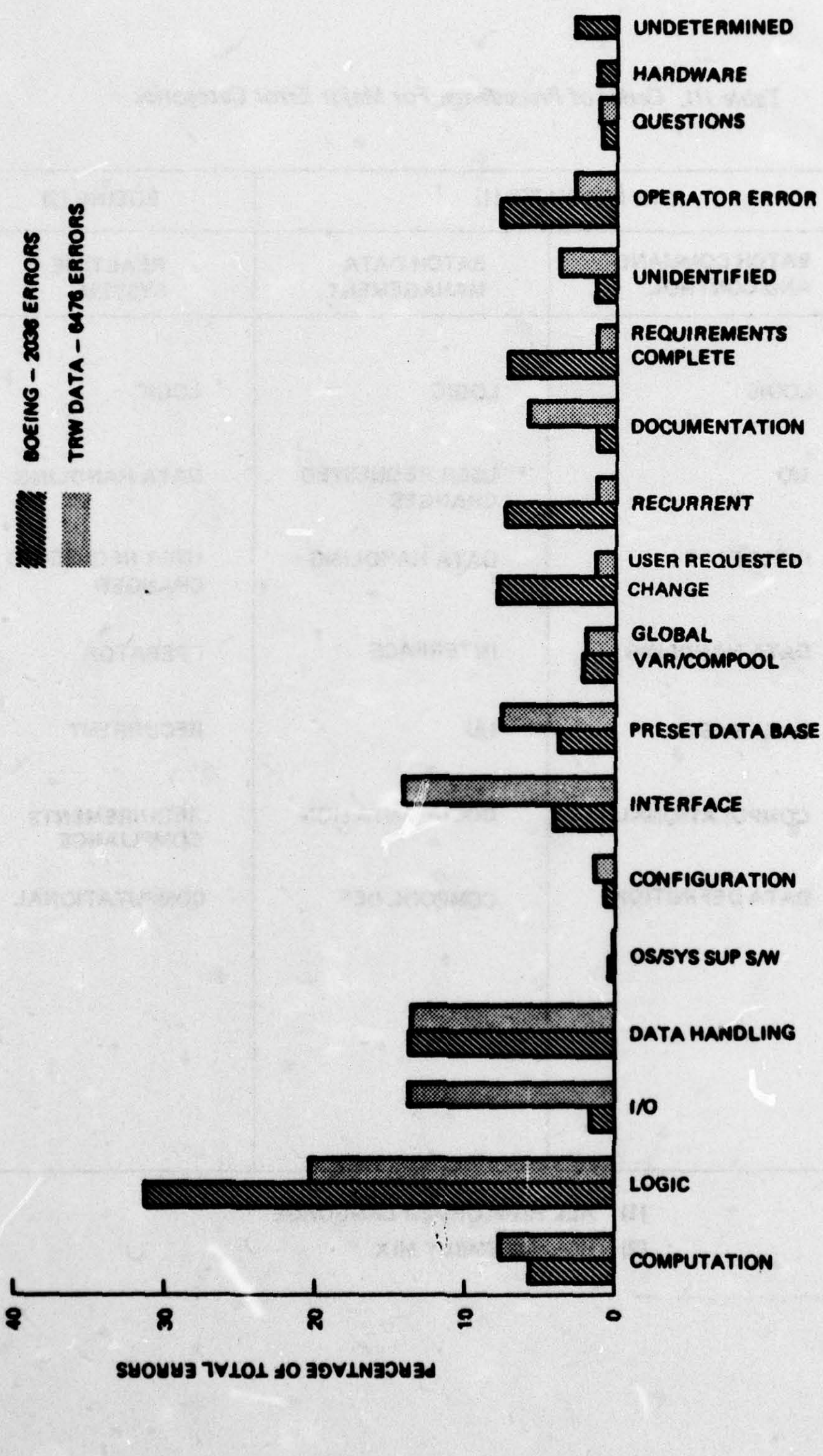

Figure 10. Boeing and TRW Error Data by Category

Table III. Order of Precedence For Major Error Categories

	TRW PRODUCTS (1)		BOEING (2)
	BATCH COMMAND AND CONTROL	BATCH DATA MANAGEMENT	REALTIME SYSTEM
1	LOGIC	LOGIC	LOGIC
2	I/O	USER REQUESTED CHANGES	DATA HANDLING
3	INTERFACE	DATA HANDLING	USER REQUESTED CHANGES
4	DATA HANDLING	INTERFACE	OPERATOR
5	DATA BASE	I/O	RECURRENT
6	COMPUTATIONAL	DOCUMENTATION	REQUIREMENTS COMPLIANCE
7	DATA DEFINITION	COMPOOL DEF	COMPUTATIONAL

(1) ALL HIGH ORDER LANGUAGE
(2) HOL/ASSEMBLY MIX

The results of the Chi-square tests (in Table IV) showed that the differences between the Boeing and TRW data and the value represented by the average of the two were significant in only the category I/O. In the case of I/O, this may be explained because there was very little external I/O going on in this Boeing system. There is a great deal of data passing between computers and peripherals but errors of this type are not described in the I/O category.

Figures 9A, 9B and 9C show the Boeing data by category and functional area. In general, there was good agreement of the results across the functional areas. The subsystem simulator software showed a higher percentage of computational errors, including mathematical modeling errors. This is consistent with its function of modeling the behavior of an airborne system. This modeling was subject to scaling, coordinate, and parameter problems and reflects the difficulty of communicating a design for a complicated simulation. It is often an iterative process, continuing until the design requirements as understood by the programmer match the actual requirements.

The subsystem simulator system and the executive system both show relatively high percentages of User Requested Changes. Both have high visibility to the users since they function as utility-like systems.

The hardware test function shows a very high percentage of Logic errors, consistent with an equipment test function. In addition, this software was the last to be developed. Unlike the other functions, which had been subjected to months of testing prior to IMCT, this function was being developed and tested just prior to and concurrent with the first IMCT period. Hence, there was not as much opportunity to eliminate errors before formal testing began and evidently many logic errors still remained.

5.2 Additional Results

5.2.1 Intermodule Error Rate Classification

Another interesting area is the relationship of errors to the modules involved in the error. It is often thought that the modules in a large piece of software are highly interrelated. One of the results of this study is the number of errors versus the number of modules involved in the error shown on Figure 11. As the data shows, the majority of errors involved only one module and do not involve errors across the interface among the modules.

It is interesting that this result tracks so well with the results found in a paper by Albert Endres (2). The comparison is shown in Table V. Endres' data was based on a study done on systems programs during a critical testing phase. His study clearly supports the notion that modules are not as interrelated as some believe. This kind of data is important to have for example, when planning maintenance efforts. If changes in one module are going to propagate through many others the impact is much greater than if the effect of changes are isolated to one module.

Table IV. Chi-Square Test Results

CATEGORY	BOEING	TRW	AVERAGE	X^2
COMPUTATIONAL	5.7	8	6.85	.39
LOGIC	32.7	20.5	26.6	2.8
I/O	1.5	14.1	7.8	10.18
DATA HANDLING	14.1	13.7	13.9	.01
OPERATING SYSTEM/SYSTEM SUPPORT SOFTWARE	.4	.08	.24	.21
CONFIGURATION	.6	1.5	1.05	.39
ROUTINE/ROUTINE INTERFACE	2.1	5.4	3.75	1.45
ROUTINE/SYSTEM SOFTWARE INTERFACE	.2	0.5	.35	.13
TAPE PROCESSING INTERFACE	.3	0.3	.3	0
USER INTERFACE	.6	7.4	4.0	5.78
DATA BASE INTERFACE	.8	.7	.75	.01
USER REQUESTED CHANGES	8.3	1.7	5.0	4.36
PRESET DATA BASE	3.5	7.6	5.5	1.51
GLOBAL VAR/COMPOOL DEF	2.4	1.8	2.1	.09
RECURRENT	7.7	1.6	4.65	4.0
DOCUMENTATION	1.4	6.0	3.7	2.86
REQUIREMENTS COMPLIANCE	7.5	.9	4.2	5.19
UNIDENTIFIED	1.6	4.0	2.8	1.03
OPERATOR	8.2	3.0	5.6	2.41
QUESTIONS	.9	.8	.85	.01

FOR AN α OF .05 a$X^2 > 5.99$ MEANS REJECTION OF THE NULL HYPOTHESIS, VIZ., THE DIFFERENCES IN THE RESULTS ARE NOT DUE TO SAMPLING

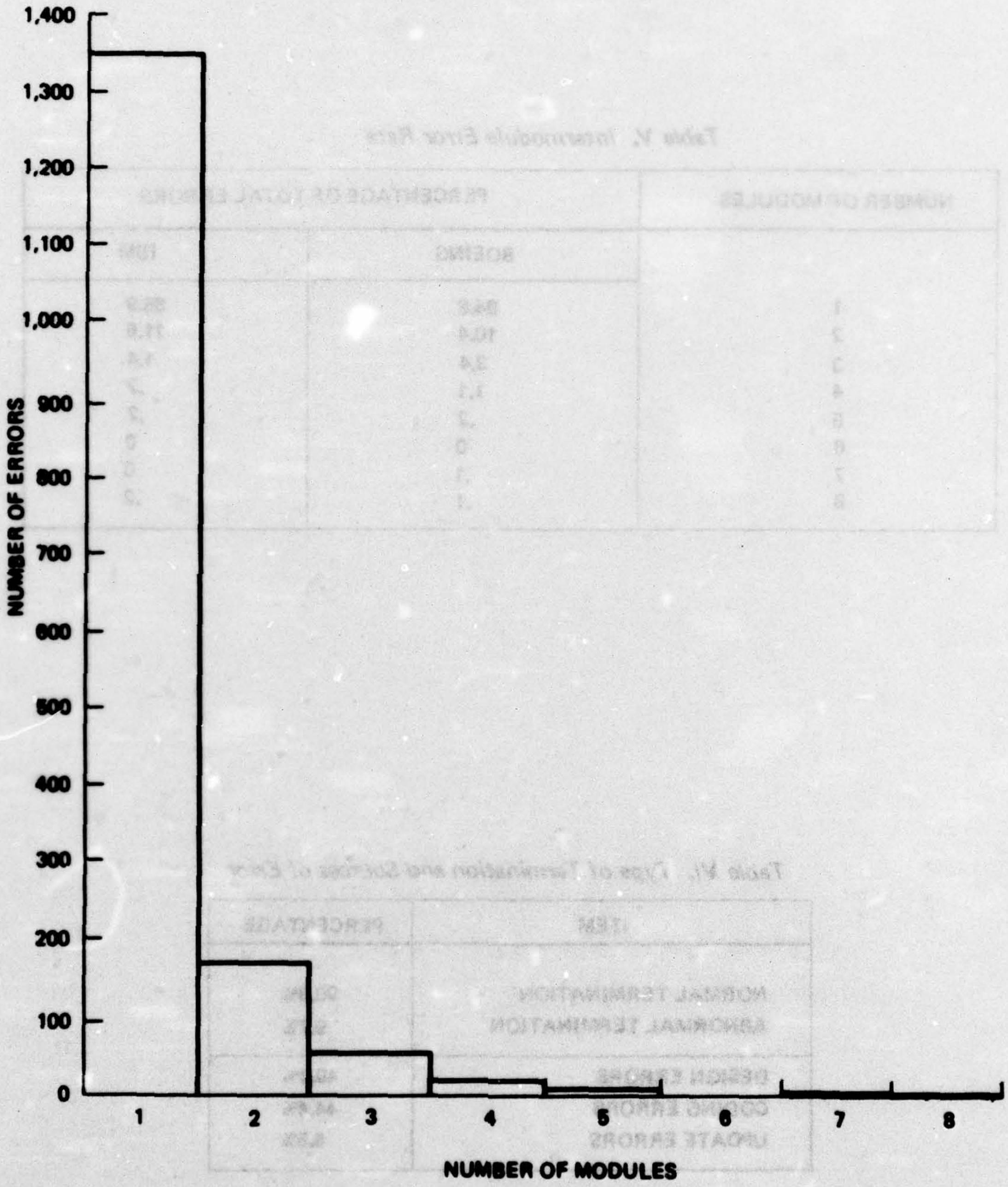

Figure 11. Intermodule Error Rate in Boeing Data

Table V. Intermodule Error Rate

NUMBER OF MODULES	PERCENTAGE OF TOTAL ERRORS	
	BOEING	IBM
1	84.8	85.9
2	10.4	11.6
3	3.4	1.4
4	1.1	.7
5	.2	.2
6	0	0
7	.1	0
8	.1	.2

Table VI. Type of Termination and Sources of Error

ITEM	PERCENTAGE
NORMAL TERMINATION	90.3%
ABNORMAL TERMINATION	9.7%
DESIGN ERRORS	49.1%
CODING ERRORS	44.4%
UPDATE ERRORS	6.5%

5.2.2 Termination and Development Classification

In addition to classification by error type, it was required in the study that all the SPRs be classified by development phase information (Design, Code, Update) and by test termination (Normal, Abnormal). The termination and development information from Boeing data is summarized in Table VI. The majority of terminations were normal. That is, the system did not crash, rather it remained operating, although incorrectly relative to the symptoms of the particular error.

The percentage of design errors was about 50%, indicating a need to support development of design tools since half the errors occur in this phase of the project. In addition, a surprisingly high 6.5% of the errors were a result of attempts to fix previous errors or update the software. Thus, the number of errors introduced by the correction process itself is nontrivial. This is an important consideration when developing reliability model assumptions.

5.2.3 Error Rate By System Functional Area

As the data was being collected it became apparent that some functions had higher error rates than others. This seemed an interesting area to pursue since, if certain functions showed themselves to be more error prone than others, it would be important information to have for future projects.

In order to make this comparison two pieces of data were needed. The first was the size of the software. The size of the software was expressed in half-words of core which would be needed to contain all the instructions and data in each functionally separate set of modules. This is exactly the data on code size tracked by the project. The second set of data was the number of errors found for each functional area. In this case errors were defined to be "real" software errors; that is, it did not include Software Problem Reports which were duplicates or problems attributed to the categories Hardware, Questions, Documentation, Operator and User Requested Changes. The ratio, number of errors/software size, was used as a measure of error rate, i.e., errors generated per core locations used. It needs to be noted here that the size of the software was expressed in core locations because several functional areas contained programs written in both HOL and assembly language. Using core locations to express code size reduces all functional areas to the same units. Table VII summarizes these results.

It is dangerous to read too much into such numbers, since it is not really possible to separate all errors involving instructions from all errors involving data. Furthermore, additional errors were found subsequent to the release of Block 1, i.e., during release of Block 2 and Block 3. Still at the coarse level the Controls and Displays function has a remarkably lower error rate. In our opinion, this is due to two factors. One, it must have been an exceptionally well thought out programming task to

Table VII. Error Rate by Functional Area

FUNCTIONAL AREA	SIZE OF SOFTWARE IN HALFWORDS	"REAL"* ERRORS	ERRORS/HALFWORDS OF CORE
SUBSYSTEM SIMULATOR	21000 INSTR 11000 DATA	240	.0075
SYSTEM SIMULATOR	16000 INSTR 16000 DATA	188	.0059
EXECUTIVE SOFTWARE	19800 INSTR 17500 DATA	195	.0052
SYSTEM FUNCTION A	8500 INSTR 9400 DATA	198	.0111
CONTROLS & DISPLAYS	25000 INSTR 7000 DATA	73	.0023
HARDWARE TEST MONITOR	30000 INSTR 25000 DATA	412	.0075
SYSTEMS FUNCTION B	22000 INSTR 11000 DATA	155	.0048

* REAL ERRORS DO NOT INCLUDE SPR'S WHICH ARE DUPLICATES OR SPR'S CATEGORIZED AS HARDWARE, QUESTIONS, DOCUMENTATION, OPERATOR, AND USER REQUESTED CHANGES.

have spawned so few errors. Second, although as a programming task it had complexities (e.g., the use of multiply-linked lists), as an engineering problem it was simpler than the other areas. Besides the programming complexities in the building of the subsystem simulator program, there is the added complexity of adequately communicating the scope of the engineering complexities to the programmer initially. This phenonemon may contribute to the higher error rate of the system function A software. It represented a sophisticated complex engineering problem.

5.2.4 Time to Close SPRs

As a last item in the survey of the data, Figure 12 shows a representation of the length of time an SPR was open at different times in the software development cycle. In all cases except the last Systems Validation Test period, a majority of SPRs were closed within a month, and over 80% by two months. This, it must be remembered, includes time to update source code, recompile and retest. The long-time errors are usually of two types: first, those not needed to be fixed until the next software release (the majority were this type); and second, those which are difficult to fix. The last Systems Validation phase was followed by a substantial time lapse until release of the next block of software (Block 2). This probably accounts for the lack of push to close most SPR's quickly during the SVT phase of Block 1.

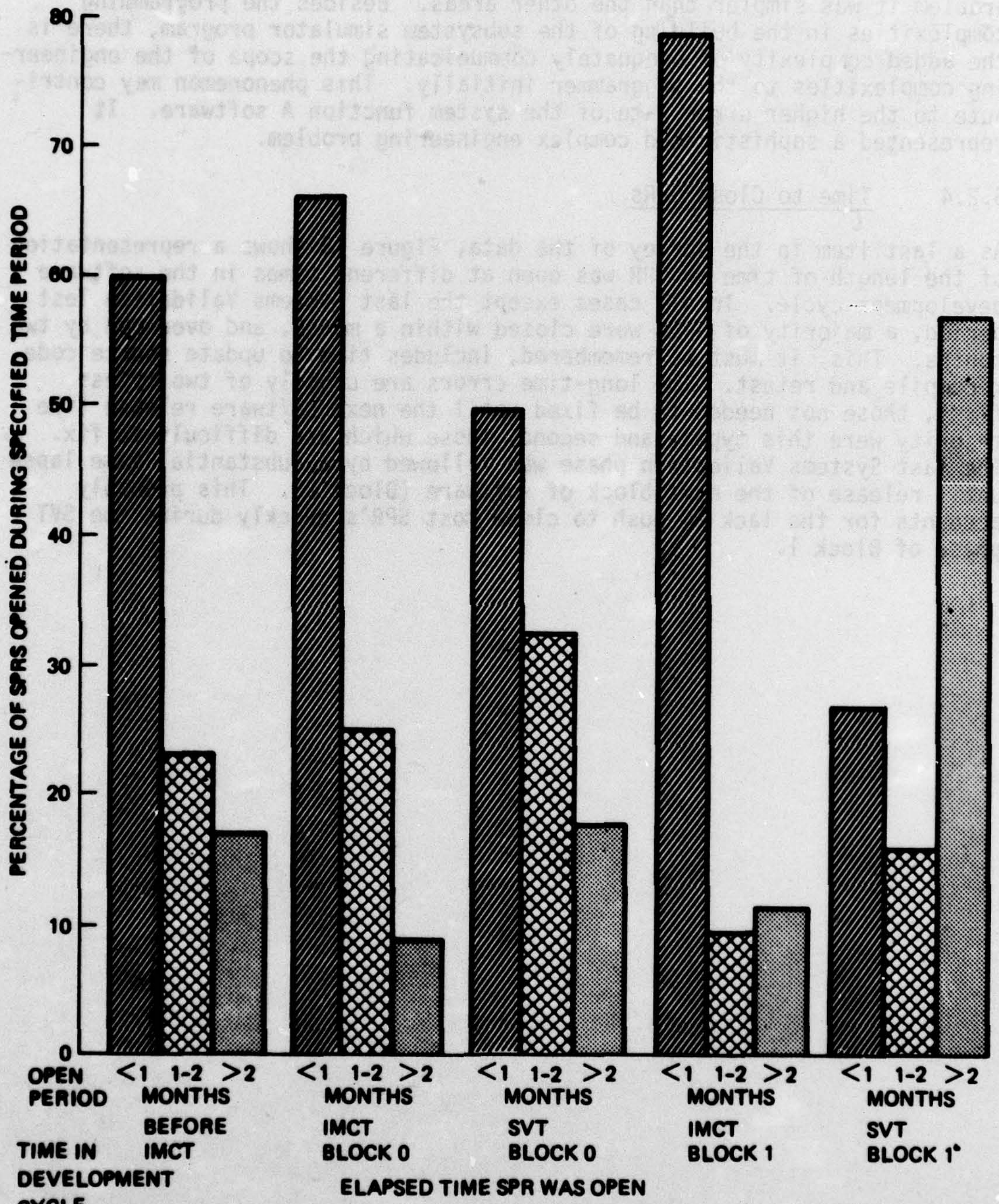

Figure 12. Time to Close Error Reports

6 CONCLUSIONS

The most significant conclusion is the affirmation that much useful data can be collected successfully in an ongoing software project. In fact, there proved to be a large amount of information available in project records which could be refined into a new, useful form.

In addition, it must be concluded that such data needs to be collected. Project people are frequently called upon to estimate change rates and coding productivity in planning a future maintenance phase. Or they may need hard information to support the planning of new software projects, including estimates of required testing time to completion, cost of testing, general information about where in the development cycle errors arise and the type of the errors. In fact, two such requests for data were made during this contract. Hence, such data collection is useful, in fact, it should be expanded in its scope.

From this data collection experience, it became apparent that there already exists a framework in which to do this type of collection - that is, the configuration management organization. This was the group which kept the records of all changes in the software, kept documentation up to date and in general was a source of much general data about the software development process. This was all done in conjunction with the job of maintaining control over the software. With a little bit of modification and addition, these functions could easily incorporate record keeping for data collection of the type required by this study.

Accepting the usefulness of such data, when and in what form should collection be done during software development? To assure the best possible results, plans for data collection should be done before the start of the project and include information collected about the early stages of software development, particularly the requirements, specifications and design processes. Time and other resources expended in all phases of the software development process should be carefully tracked. Such planning before the start of the project assures that the data will be in the form needed. Often data cannot be reconstructed later for lack of some small items which could easily have been collected, if anticipated in the planning phase.

Second, a well planned software problem report is an essential. It should contain all the basic information of interest about the discovery and correction of an error. It is possible to collect a great deal of information with one very complete report sheet. The software problem reports used in this study were remarkably complete. What they needed was enforcement of completion and an agreement on the interpretation of some items.

The list of possible errors should be short. As mentioned previously, our opinion is that it should be only as long as a person can easily house in their mind and apply from memory. In this respect, the new shorter list

in the TRW report (1) referenced earlier seems an improvement. It is shorter, the essential categories have been kept, and the out-lying and non-coding related errors have been grouped together. However, we would make a separate category for interface errors in an embedded computer system, i.e., errors occurring between intersystem elements.

Information collected should include data on the tools used to discover the error if any special ones were used. This information is useful in evaluating the effectiveness of software validation and verification tools.

The occurrence of errors should be attributed to requirements, design, coding, update, and possibly maintenance phases. It is necessary to carefully define these terms however. We suggest this be done in terms of documentation. Inherent in all of this is the need to prepare and motivate the programmers and test personnel adequately. The forms should be reviewed item by item to assure understanding. Documentation on the categories should be available. Last, the need for such data should be clearly explained. In conjunction with this, data should be made available as it is developed for interested parties to peruse.

One or two people should be assigned the responsibility of insuring that forms are properly filled out and other necessary data supplied. A number of forms which are not filled out or which are filled out improperly will impair the results.

Based on our experience in this study, it is recommended that equipment hours be the measure of time to discovery of an error and time to fix an error. This applies to systems such as the one covered in this study, where the computer is embedded in the system and the software runs as part of the operation of the total system.

Other information which could be collected and should be of help in interpreting data is statistics on the code itself. This should include length of code, number and type of input items, number and type of output items, number of branches, and some general characteristics of the code (e.g., list processing, computational, error checking, etc.). Moreover, programmers should be instructed to collect accurate records of time to do actual coding, time to do initial debug, desk hours spent finding errors, and the time spend doing documentation.

The opportunity for data collection in a project is great. The payoff of careful collection is greater yet - a source of information for planning and improving future software development projects based on past experiences and a basis for evaluating software development techniques realistically.

7 REFERENCES

1. Thayer, T. et al., "Software Reliability Study", RADC-TR-76-238, Final Technical Report, August 1976. AD#A030798.

2. Albert Endres, "An Analysis of Errors and Their Causes in Systems Programs," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June, 1975.

8 APPENDIX A

8.1 Error Categories

In this appendix, a tabular list of error categories is provided.

ERROR CATEGORIES

CATEGORY ID	CATEGORIES
AA000	COMPUTATIONAL ERRORS
AA010	TOTAL NUMBER OF ENTRIES COMPUTED INCORRECTLY
AA020	PHYSICAL OR LOGICAL ENTRY NUMBER COMPUTED INCORRECTLY
AA040	WRONG EQUATION OR CONVENTION USED
AA041	MATHEMATICAL MODELING PROBLEM
AA050	RESULTS OF ARITHMETIC CALCULATION INACCURATE/NOT AS EXPECTED
AA060	MIXED MODE ARITHMETIC ERROR
AA070	TIME CALCULATION ERROR
AA071	TIME CONVERSION ERROR
AA072	TIME TRUNCATION/ROUNDING ERROR
AA080	SIGN CONVENTION ERROR
AA090	UNITS CONVERSION ERROR
AA100	VECTOR CALCULATION ERROR
AA110	CALCULATION FAILS TO CONVERGE
AA120	QUANTIZATION/TRUNCATION ERROR
BB000	LOGIC ERRORS
BB010	LIMIT DETERMINATION ERROR
BB020	WRONG LOGIC BRANCH TAKEN
BB030	LOOP EXITED ON WRONG CYCLE
BB040	INCOMPLETE PROCESSING
BB050	ENDLESS LOOP DURING ROUTINE OPERATION
BB060	MISSING LOGIC OR CONDITION TEST
BB061	INDEX NOT CHECKED
BB062	FLAG OR SPECIFIC DATA VALUE NOT TESTED
BB070	INCORRECT LOGIC
BB080	SEQUENCE OF ACTIVITIES WRONG
BB090	FILTERING ERROR
BB100	STATUS CHECK/PROPAGATION ERROR
BB110	ITERATION STEP SIZE INCORRECTLY DETERMINED
BB120	LOGICAL CODE PRODUCED WRONG RESULTS
BB130	LOGIC ON WRONG ROUTINE
BB140	PHYSICAL CHARACTERISTICS OF PROBLEM TO BE SOLVED, OVERLOOKED, OR MISUNDERSTOOD
BB150	LOGIC NEEDLESSLY COMPLEX
BB160	INEFFICIENT LOGIC
BB170	EXCESSIVE LOGIC
BB180	STORAGE REFERENCE ERROR (SOFTWARE PROBLEM)

ERROR CATEGORIES

CATEGORY ID	CATEGORIES
CC000	I/O ERRORS
CC010	MISSING OUTPUT
CC020	OUTPUT MISSING DATA ENTRIES
CC030	ERROR MESSAGE NOT OUTPUT
CC040	ERROR MESSAGE GARBLED
CC050	OUTPUT OR ERROR MESSAGE NOT COMPATIBLE WITH DESIGN DOCUMENTATION (INCLUDING GARBLED OUTPUT)
CC060	MISLEADING OR INACCURATE ERROR MESSAGE TEXT
CC070	OUTPUT FORMAT ERROR (INCLUDING WRONG LOCATION)
CC080	DUPLICATE OR EXCESSIVE OUTPUT
CC090	OUTPUT FIELD SIZE INADEQUATE
CC100	DEBUG OUTPUT PROBLEM (RELATIVE TO DESIGN DOCUMENTATION)
CC101	LACK OF DEBUG OUTPUT
CC102	TOO MUCH DEBUG
CC110	HEADER OUTPUT PROBLEM
CC120	OUTPUT TAPE FORMAT ERROR
CC130	OUTPUT CARD FORMAT ERROR
CC140	ERROR IN PRINTER CONTROL
CC150	LINE COUNT/PAGE EJECT ERROR
CC160	NEEDED OUTPUT NOT PROVIDED IN DESIGN
CC161	INSUFFICIENT OUTPUT OPTIONS
DD000	DATA HANDLING ERRORS
DD010	VALID INPUT DATA IMPROPERLY SET/USED
DD020	DATA WRITTEN IN OR READ FROM WRONG DISK LOCATION
DD030	DATA LOST/NOT STORED
DD040	DATA, INDEX OR FLAG NOT SET OR SET/INITIALIZED INCORRECTLY
DD041	NUMBER OF ENTRIES SET INCORRECTLY
DD050	DATA, INDEX OR FLAG MODIFIED OR UPDATED INCORRECTLY
DD051	NUMBER OF ENTRIES UPDATED INCORRECTLY
DD060	EXTRANEOUS ENTRIES GENERATED (TABLE, ARRAY, ETC.)
DD070	BIT MANIPULATION ERROR
DD071	ERROR USING BIT MODIFIER
DD080	FLOATING POINT/INTEGER CONVERSION ERROR
DD090	INTERNAL VARIABLE ERROR (DEFINITION OR SET/USE)
DD100	DATA PACKING/UNPACKING ERROR
DD110	ROUTINE LOOKING FOR DATA IN NON-EXISTENT RECORD
DD120	BOUNDS VIOLATION
DD130	DATA CHAINING ERROR
DD140	DATA OVERLOW OR OVERFLOW PROCESSING ERROR
DD150	READ ERROR
DD151	ALL AVAILABLE DATA NOT READ
DD160	LONG LITERAL PROCESSING ERROR
DD170	SORT ERROR
DD180	OVERLAY ERROR
DD190	SUBSCRIPTING CONVENTION ERROR
DD200	DOUBLE BUFFERING ERROR

ERROR CATEGORIES

CATEGORY ID	CATEGORIES
EE000	OPERATING SYSTEM/SYSTEM SUPPORT SOFTWARE ERRORS
EE010	JOVIAL PRODUCES ERRONEOUS MACHINE CODE
EE020	OS MISSING NEEDED CAPABILITY
FF000	CONFIGURATION ERRORS
FF010	COMPILATION ERROR
FF011	SEGMENTATION PROBLEM
FF020	ILLEGAL INSTRUCTION
FF030	UNEXPLAINABLE PROGRAM HALT
GG000	ROUTINE/ROUTINE INTERFACE ERRORS
GG010	ROUTINE PASSING INCORRECT AMOUNT OF DATA (INSUFFICIENT OR TOO MUCH)
GG020	ROUTINE PASSING WRONG PARAMETERS OR UNITS
GG030	ROUTINE EXPECTING WRONG PARAMETERS
GG040	ROUTINE FAILS TO USE AVAILABLE DATA
GG050	ROUTINE SENSITIVE TO INPUT DATA ORDER
GG060	CALLING SEQUENCE OR ROUTINE/ROUTINE INITIALIZATION ERROR
GG070	ROUTINES COMMUNICATING THROUGH WRONG DATA BLOCK
GG080	ROUTINE USED OUTSIDE DESIGN LIMITATION
GG090	ROUTINE WON'T LOAD (ROUTINE INCOMPATIBILITY)
GG100	ROUTINE OVERFLOWS CORE WHEN LOADED
HH000	ROUTINE/SYSTEM SOFTWARE INTERFACE ERRORS
HH010	OS INTERFACE ERROR (CALLING SEQUENCE OR INIALIZATION)
HH020	ROUTINE USES EXISTING SYSTEM SUPPORT SOFTWARE INCORRECTLY
HH030	ROUTINE USES SENSE/JUMP SWITCH IMPROPERLY
II000	TAPE PROCESSING INTERFACE ERROR
II010	TAPE UNIT EQUIPMENT CHECK NOT MADE
II020	ROUTINE FAILS TO READ CONTINUATION TAPE
II030	ROUTINE FAILS TO UNLOAD TAPE AFTER COMPLETION
II040	ERRONEOUS INPUT TAPE FORMAT

ERROR CATEGORIES

CATEGORY ID	CATEGORIES
JJ000	USER INTERFACE ERRORS
JJ010	OPERATIONS REQUEST OR DATA CARD/ROUTINE INCOMPATABILITY
JJ020	MULTIPLE PHYSICAL CARD/LOGICAL CARD PROCESSING ERROR
JJ030	INPUT DATA INTERPRETED INCORRECTLY BY ROUTINE
JJ040	VALID INPUT DATA REJECTED OR NOT USED BY ROUTINE
JJ050	INPUT DATA REJECTED BUT USED
JJ060	INPUT DATA READ BUT NOT USED
JJ070	ILLEGAL INPUT DATA ACCEPTED AND PROCESSED
JJ080	LEGAL INPUT DATA PROCESSED INCORRECTLY
JJ090	POOR DESIGN IN OPERATOR INTERFACE
JJ100	INADEQUATE INTERRUPT AND START CAPABILITY
KK000	DATA BASE INTERFACE ERRORS
KK010	ROUTINE/DATA BASE INCOMPATIBILITY
KK011	UNCOORDINATED USE OF DATA ELEMENTS BY MORE THAN ONE USER
LL000	USER REQUESTED CHANGES
LL010	SIMPLIFIED INTERFACE AND/OR CONVENIENCE
LL020	NEW AND/OR ENHANCED FUNCTIONS
LL021	CPU
LL022	DISK
LL023	TAPE
LL024	I/O
LL025	CORE
LL030	SECURITY
LL040	NEW HARDWARE/OS CAPABILITY
LL050	INSTRUMENTATION
LL060	CAPACITY
LL070	DATA BASE MANAGEMENT AND INTEGRITY
LL080	EXTERNAL PROGRAM INTERFACE
MM000	PRESET DATA BASE ERRORS
MM010	DATA OR OPERATIONS REQUEST CARD DESCRIPTIONS
MM020	ERROR MESSAGE TEXT
MM030	NOMINAL, DEFAULT, LEGAL, MAX/MIN VALUES
MM040	PHYSICAL CONSTANTS AND MODELING PARAMETERS
MM041	EPHEMERIS PARAMETERS
MM050	DICTIONARY (BIT STRING) PARAMETERS
MM060	MISSING DATA BASE SETTINGS

ERROR CATEGORIES

CATETORY ID	CATEGORIES
NN000	GLOBAL VARIABLE/COMPOOL DEFINITION ERRORS
NN010	ITEMS IN WRONG LOCATION (WRONG DATA BLOCK)
NN011	DEFINITION SEQUENCE ERROR
NN020	DATA DEFINITION ERROR
NN021	TABLE DEFINITION INCORRECT
NN030	LENGTH OF DEFINITION INCORRECT
NN040	COMMENTS ERROR
NN050	DELETE UNNEEDED DEFINITIONS
PP000	RECURRENT ERRORS
PP010	PROBLEM REPORT REOPENED
PP020	PROBLEM REPORT A DUPLICATE OF PREVIOUS REPORT
QQ000	DOCUMENTATION ERRORS
QQ010	ROUTINE LIMITATION
QQ020	OPERATING PROCEDURES
QQ030	DIFFERENCE BETWEEN FLOW CHART AND CODE
QQ040	TAPE FORMAT
QQ050	DATA CARD/OPERATION REQUEST CARD FORMAT
QQ060	ERROR MESSAGE
QQ070	ROUTINE'S FUNCTIONAL DESCRIPTION
QQ080	OUTPUT FORMAT
QQ090	DOCUMENTATION NOT CLEAR/NOT COMPLETE
QQ100	TEST CASE DOCUMENTATION
QQ110	OPERATING SYSTEM DOCUMENTATION
QQ120	TYPO/EDITORIAL ERROR/COSMETIC CHANGE
RR000	REQUIREMENTS COMPLIANCE ERRORS
RR010	EXCESSIVE RUN TIME
RR020	REQUIRED CAPABILITY OVERLOOKED OR NOT DELIVERED AT TIME OF REPORT

ERROR CATEGORIES

CATEGORY ID	CATEGORIES
SS000	UNIDENTIFIED ERRORS
TT000	OPERATOR ERROR
TT010	TEST EXECUTION ERROR
TT020	ROUTINE COMPILED AGAINST WRONG COMPOOL/MASTER COMMON
TT030	WRONG DATA BASE USED
TT040	WRONG MASTER CONFIGURATION USED
TT050	WRONG TAPE(S) USED
UU000	QUESTIONS
UU010	DATA BASE
UU020	MASTER CONFIGURATION
UU030	ROUTINE
V	HARDWARE
X	NON-REPRODUCIBLE

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg-K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
$1\ 000\ 000\ 000\ 000 = 10^{12}$	tera	T
$1\ 000\ 000\ 000 = 10^{9}$	giga	G
$1\ 000\ 000 = 10^{6}$	mega	M
$1\ 000 = 10^{3}$	kilo	k
$100 = 10^{2}$	hecto*	h
$10 = 10^{1}$	deka*	da
$0.1 = 10^{-1}$	deci*	d
$0.01 = 10^{-2}$	centi*	c
$0.001 = 10^{-3}$	milli	m
$0.000\ 001 = 10^{-6}$	micro	μ
$0.000\ 000\ 001 = 10^{-9}$	nano	n
$0.000\ 000\ 000\ 001 = 10^{-12}$	pico	p
$0.000\ 000\ 000\ 000\ 001 = 10^{-15}$	femto	f
$0.000\ 000\ 000\ 000\ 000\ 001 = 10^{-18}$	atto	a

** To be avoided where possible.*

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

AMERICAN REVOLUTION BICENTENNIAL 1776-1976

☆U.S. GOVERNMENT PRINTING OFFICE: 1977-714-025/324